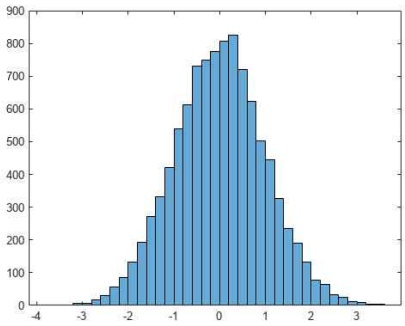
1. 기초 세팅



1.1 Histogram이란

Histogram이란 데이터의 분포를 그래프로 표현한 것이다. 특정 데이터 또는 데이터 범위의 발생 빈도를 나타내며, 이미지의 픽셀별 밝기값의 분포를 시각화한다. Histogram을 통해 이미지 또는 영상의 밝기나 대비를 파악할 수 있으며, 특정 부분의 픽셀별 밝기값의 분포 또한 알 수 있다.

1.2 Python을 사용한 이유

Python은 이미지를 처리하는데 사용되는 다양한 라이브러리를 제공하기에 Python을 사용하였다. PIL, matplotlib, numpy, scipy 라이브러리를 사용한 뒤, Canny filter을 구현하기 위해 OpenCV (cv2) 라이브러리를 사용하였다. PIL 라이브러리는 이미지를 로드하고 저장하는데 사용하였으며, matplotlib 라이브러리는 영상처리한 이미지나 Histogram을 시각화하기 위해 사용하였다. numpy 라이브러리는 이미지의 픽셀별 밝기값을 연산하거나 배열 형태로 정리해 Smoothing 또는 Edge Detection을 하기 위해 사용하셨다. scipy 라이브러리 또한 가우시안 필터 등을 적용시키는데 사용하였다.

2. 얼굴 원영상과 Histogram

컬러 이미지는 처리하기 어렵기 때문에 컬러 이미지를 흑백으로 변환하여 사용하였다. 코드는 아래와 같다.

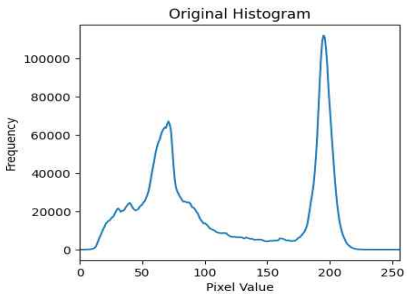
```
from PIL import Image  //이미지를 불러와 처리하는데 사용
import matplotlib.pyplot as plt //처리한 이미지를 시각화하기 위해 사용

image = Image.open('faceimage.jpg') //'faceimage.jpg'로 저장된 얼굴 원영상 불러오기
bw_image = image.convert('L') //흑백 이미지로 변환
bw_image.save('bw_image.jpg') //흑백 이미지 저장

plt.subplot(1, 1, 1)  //얼굴 원영상의 Histogram 출력
plt.title('Original Histogram') //Histogram의 제목
plt.xlabel('Pixel Value')  //Histogram의 x축 제목
plt.ylabel('Frequency')  //Historam의 y축 제목
plt.plot(bw_image.histogram())  //Histogram 그리기
plt.xlim([0, 256])  //x축의 범위 설정
plt.show()
```

코드를 실행하면 폴더에 아래와 같은 'bw_image.jpg' 파일이 생성되고, Histogram 결과가 출력된다.

<faceimage.jpg>

<bw_image.jpg>

<Histogram 결과>



3. 영상을 어둡게 또는 밝게 변환한 결과 영상과 Histogram

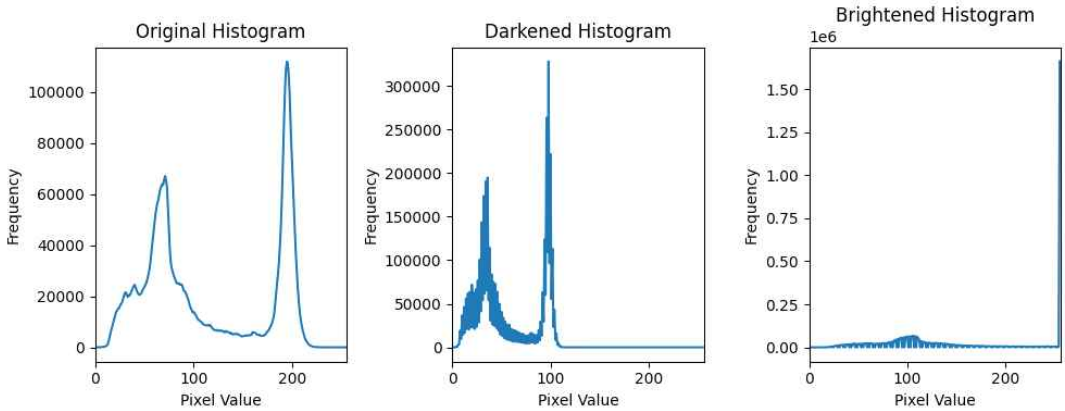
영상을 어둡게 또는 밝게 변환하기 위해 세 가지 방법을 사용해 보았다.
첫 번째로, 픽셀의 밝기값에 0.5 혹은 1.5를 곱하는 방식을 사용해 보았다. 코드와 처리한 영상, Histogram은 다음과 같다.

```
darkened_image = Image.eval(bw_image, lambda x: x*0.5)  //픽셀 값을 절반으로 줄여 어둡게 만들기
darkened_histogram = darkened_image.histogram()  //어둡게 만든 영상의 Histogram
darkened_image.save('darkened_image.jpg')  //어둡게 만든 영상 저장

brightened_image = Image.eval(bw_image, lambda x: x*1.5)  //픽셀 값을 1.5배 곱해 밝게 만들기
brightened_histogram = brightened_image.histogram()  //밝게 만든 영상의 Histogram
brightened_image.save('brightened_image.jpg')  //밝게 만든 영상 저장
```

<얼굴 원영상>

<어둡게(x0.5)처리한 영상>

<밝게(x1.5)처리한 영상>



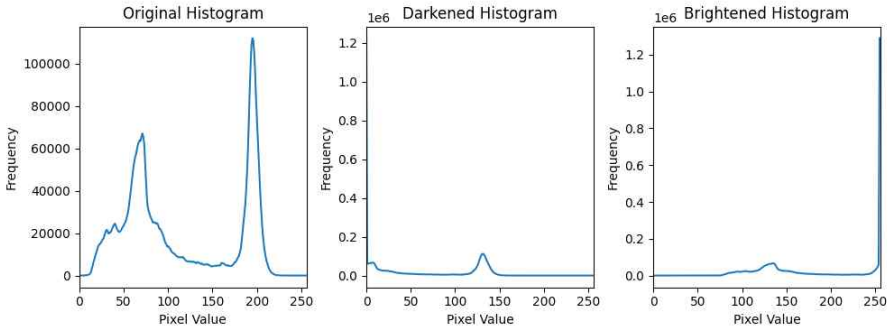
두 번째로, 픽셀의 밝기값에 64를 더하거나 빼보았다. 코드에서 붉게 표시한 부분만 변경되었다. 코드와 처리한 영상, Histogram은 다음과 같다.

```
darkened_image = Image.eval(bw_image, lambda x: x-64)  //픽셀 값을 절반으로 줄여 어둡게 만들기
darkened_histogram = darkened_image.histogram()  //어둡게 만든 영상의 Histogram
darkened_image.save('darkened_image.jpg')  //어둡게 만든 영상 저장

brightened_image = Image.eval(bw_image, lambda x: x+64)  //픽셀 값을 1.5배 곱해 밝게 만들기
brightened_histogram = brightened_image.histogram()  //밝게 만든 영상의 Histogram
brightened_image.save('brightened_image.jpg')  //밝게 만든 영상 저장
```

<얼굴 원영상>

<어둡게(-64)처리한 영상>

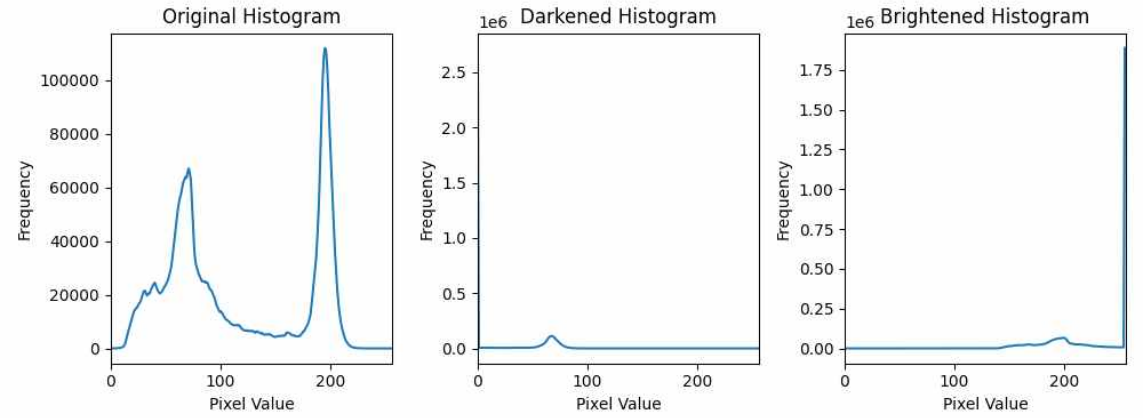
<밝게(+64)처리한 영상>



마지막으로, 픽셀의 밝기값에 128을 더하거나 빼보았다. 마찬가지로 코드에서 붉게 표시한 부분만 변경되었다. 코드와 처리한 영상, Histogram은 다음과 같다.

```
darkened_image = Image.eval(bw_image, lambda x: x-64)  //픽셀 값을 절반으로 줄여 어둡게 만들기
darkened_histogram = darkened_image.histogram()  //어둡게 만든 영상의 Histogram
darkened_image.save('darkened_image.jpg')  //어둡게 만든 영상 저장

brightened_image = Image.eval(bw_image, lambda x: x+64)  //픽셀 값을 1.5배 곱해 밝게 만들기
brightened_histogram = brightened_image.histogram()  //밝게 만든 영상의 Histogram
brightened_image.save('brightened_image.jpg')  //밝게 만든 영상 저장
```

<얼굴 원영상> <어둡게(-128)처리한 영상> <밝게(+128)처리한 영상>



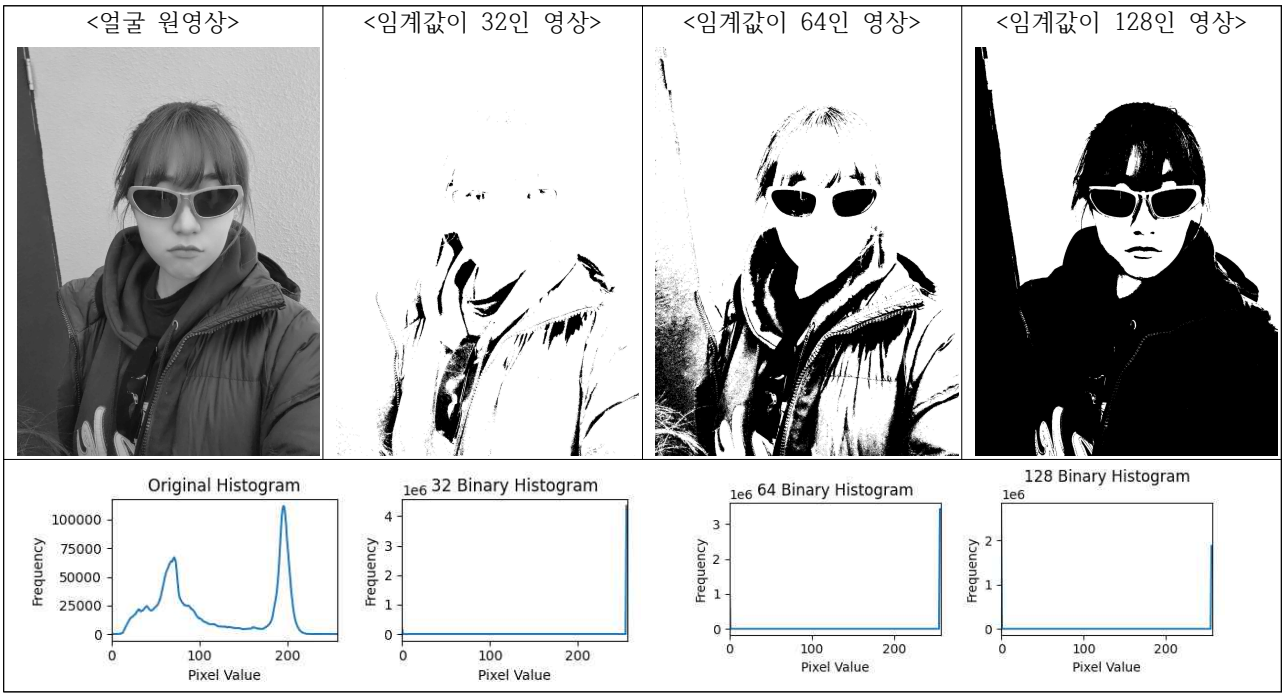
픽셀의 밝기값에 0.5배 혹은 1.5배를 곱하면 밝기차가 또렷해지며 어두운 부분은 더 어두워지고, 밝은 부분은 더 밝아짐을 알 수 있다. 반면에 64나 128을 더하거나 뺀 사진은 전체적으로 밝아지거나 어두워진 것을 알 수 있다.

4. 이진화 결과 영상

이진화란 이미지에서 특정 임계값(Threshold값)을 기준으로 픽셀의 값은 2개의 값으로 구분해 처리하는 것을 의미한다. Gray스케일 이미지에서는 임계값보다 작은 값은 0 또는 검은색으로, 임계값보다 큰 값은 255 또는 흰색으로 변환하였다. 이진화처리를 하는 코드는 아래와 같다.

```
threshold = 32  // 임계값 설정
binary32_image = bw_image.point(lambda p: 0 if p < threshold else 255, '1')
    // 임계값보다 작은 값은 0, 큰 값은 255로 변환
binary32_histogram = binary32_image.histogram()  //이진화 영상의 Histogram
binary32_image.save('binary_image32.jpg') //이진화 영상 저장
```

위 코드에서 붉게 표시한 부분만 64, 128로 바꾼 뒤, 변수명과 파일명을 다르게 저장하면 임계값을 32, 64, 128로 설정해 이진화 처리한 영상을 얻을 수 있다. 각각의 영상과 Histogram은 아래와 같다.



5. Histogram Stretching 결과 영상과 Histogram

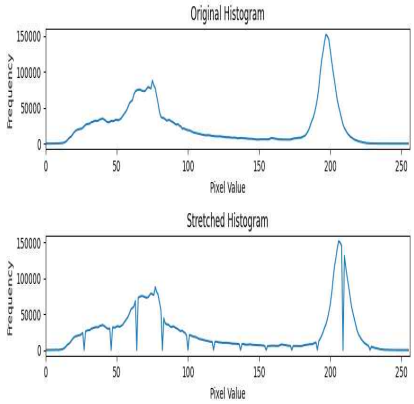
Histogram Stretching이란 이미지의 Histogram의 간격을 넓혀 이미지를 보다 또렷하게 하는 것이다. 이미지의 픽셀 밝기값 중 최대와 최소값을 찾은 뒤, $g(x,y) = \frac{255}{f_{\max} - f_{\min}}[f(x,y) - f_{\min}]$ 공식을 적용해 Stretching하였다. 코드는 아래와 같다.

```
min_val, max_val = bw_image.getextrema()  //최소값과 최대값 찾기
stretched_image = Image.eval(bw_image, lambda x: (x - min_val) * 255 / (max_val - min_val) if
max_val != min_val else 0)  //Histogram Stretching (공식 적용)
stretched_histogram = stretched_image.histogram()  //Stretching한 Histogram
stretched_image.save('stretched_image.jpg')  //Histogram Stretching한 영상 저장
```

얼굴 원영상과 Histogram Stretching한 영상, Histogram은 아래와 같다. 영상을 비교했을 때, 원영상보다 밝기차가 더 대비되는 것을 알 수 있다.

<얼굴 원영상>

<<Histogram Stretching한 영상>



6. Histogram Equalization 결과 영상과 Histogram

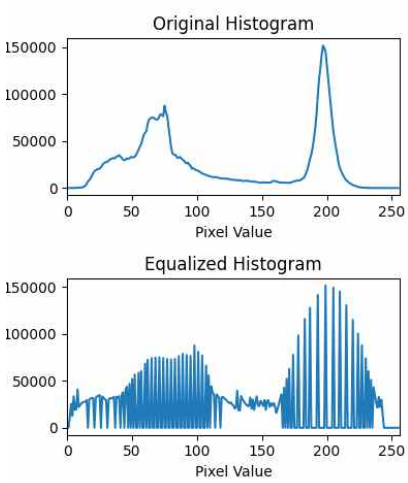
Histogram Equalization은 앞선 Histogram Stretching과 마찬가지로 이미지의 밝기와 대비를 개선하는 방법이지만, Histogram Stretching은 이미지의 Histogram이 한 쪽으로 치우져진 경우에 사용하면 Histogram을 균일하게 분포시켜 이미지의 밝기와 대비를 개선시킨다. 반면, Histogram Equalization은 각 밝기값의 발생 빈도를 계산해 재지정함으로써 Uiform한 Histogram을 얻는다. 이 때 사용되는 공식이 $g^* = \partial \left[\frac{g_{\max} - g_{\min}}{NM} \sum_{f_{\min}}^{f^*} H(f) \right] + g_{\min}$ 이다. 여기서 NM은 픽셀의 총 개수이다. Histogram Equalization을 위한 코드는 아래와 같다.

```
width, height = bw_image.size //이미지 크기 구하기
total_pixels = width * height  //NM 계산

histogram = bw_image.histogram()  //원영상 Histogram 구하기
cdf = [sum(histogram[:i + 1]) for i in range(len(histogram))]  //누적 분포 함수 계산
g_min, g_max = min(bw_image.getdata()), max(bw_image.getdata())  //픽셀의 최대값과 최소값 구하기

equalized_pixels = [round(((g_max - g_min) * cdf[pixel]) / total_pixels) + g_min for pixel in bw_image.getdata()]  //Histogram Equalization 공식 적용
equalized_image = Image.new('L', (width, height))
equalized_image.putdata(equalized_pixels)

equalized_histogram = equalized_image.histogram()  //Histogram Equalization한 영상의 Histogram 구하기
equalized_image.save('equalized_image.jpg')  //Histogram Equalization한 영상 저장
```

위 코드를 실행한 결과를 원 영상과 비교해보면 다음과 같다.

<얼굴 원영상>

<Histogram Equalization한 영상>



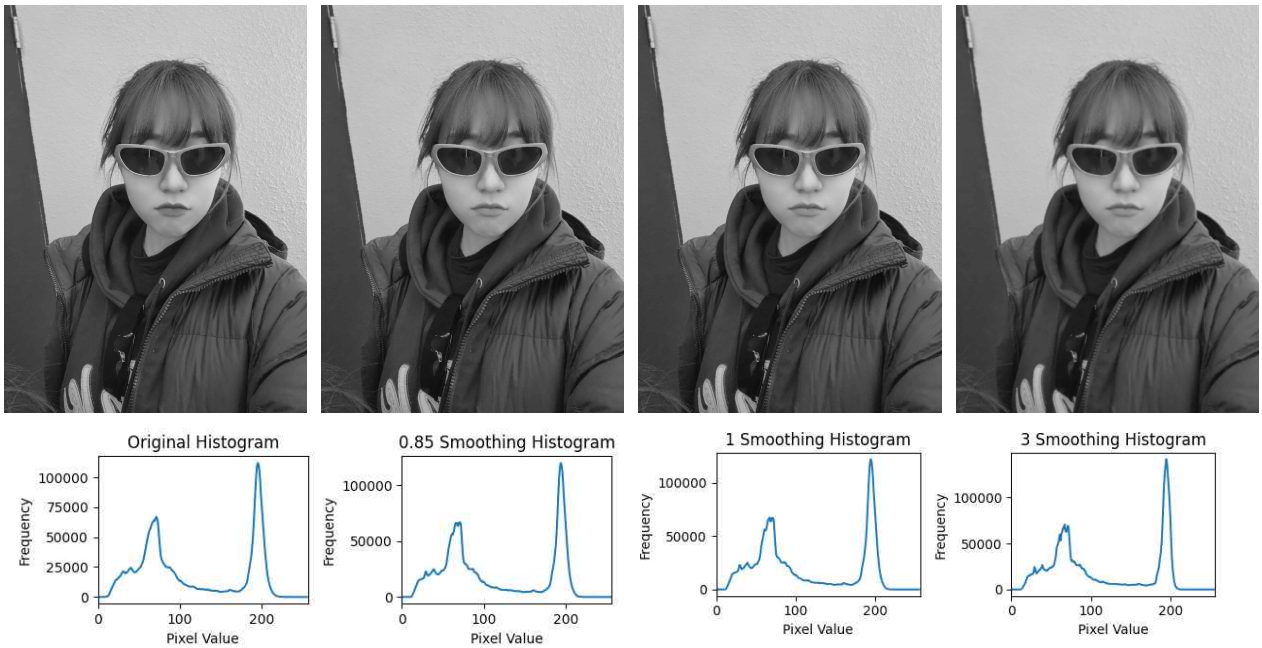
원영상과 Histogram Equalization한 영상의 차이를 보면, 우선 Equalization한 영상의 Histogram이 보다 균일하게 분포되어 있는 것을 볼 수 있다. 또, 밝은 부분과 어두운 부분의 차이가 줄어들었으며, 이미지가 더 선명해지고 세부적인 부분이 더 잘 보인다. 원영상의 잔머리나 패딩의 주름이 Equalization한 영상에서는 더 선명하게 보이는 것을 알 수 있다.

7. Gaussian Smoothing Filter 결과 영상과 Histogram

Gaussian Smoothing Filter는 이미지의 노이즈를 줄이기 위해 사용되는 영상 처리 기법이다. 이미지의 픽셀 밝기값을 평균화해 노이즈를 줄이지만, 단점으로는 이미지의 선명도가 감소할 수도 있다는 점이 있다. Gaussian Smoothing Filter은 sigma(표준편차)를 지정함으로써 정도가 편한다. sigma값이 클수록 더 많이 픽셀이 평균화되어 노이즈가 더 많이 줄어든다. Gaussian Smoothing Filter을 보다 편리하고 정확하게 적용하기 위해 Python의 scipy 라이브러리의 ndimage에서 gaussian_filter을 적용시켜 결과 영상을 출력하였다. sigma값은 0.85, 1, 3의 3가지로 적용시켜 보았으며, 코드와 결과 영상은 아래와 같다. 코드 중 붉게 표시한 부분이 sigma값을 설정하는 부분이며, 0.85를 1과 3으로 변환시키며 결과 영상을 구해보았다.

```
img_array = np.array(bw_image) //이미지 데이터를 NumPy 배열로 변환

smoothing85_array = ndi.gaussian_filter(img_array, sigma=0.85)  //Gaussian Filter 적용
smoothing85_image = Image.fromarray(smoothing85_array.astype(np.uint8)) //NumPy 배열을 이미지로 변환하여 저장
smoothing85_image.save('0.85 smoothing_image.jpg')

smoothing85_histogram = smoothing85_image.histogram()  //Gaussian Filter 적용한 영상의 Histogram
```

<얼굴원영상> <Sigma=0.85> <Sigma=1> <Sigma=3>



영상을 보면 sigma값이 높아질수록 이미지가 더 부드럽게 편하는 것을 알 수 있다. 벽의 재질이 원영상에서는 올록볼록하지만 sigma값이 3인 영상에서는 일반 매끄러운 벽처럼 보인다.

8. Sobel Gradient operator 결과 영상과 Histogram

Gradient operator 중 Sobel방법은 이미지의 경계(edge)를 검출하기 위해 사용되는 영상 처리 기법이다. 이미지의 픽셀 밝기값을 미분해 이미지의 경계에서 발생하는 밝기의 급격한 변화를 검출한다.

Sobel 방법에서 x방향으로는 [-1 0 1] 필터가, y 방향으로는 [1 2 1] 필터가 적용된다.

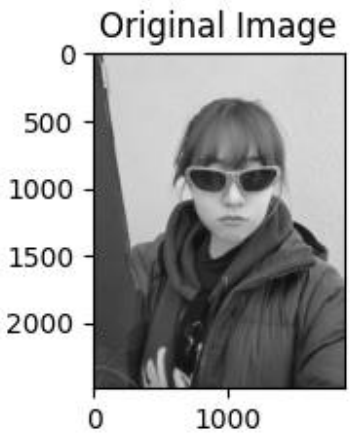
[-2 0 2] [0 0 0]

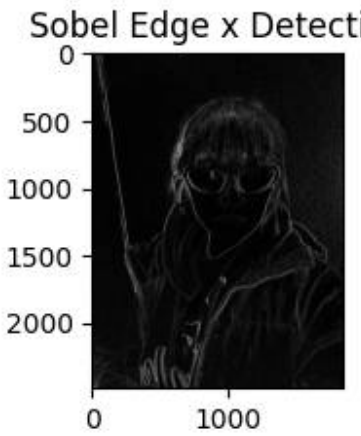
[-1 0 1] [-1 -2 -1]

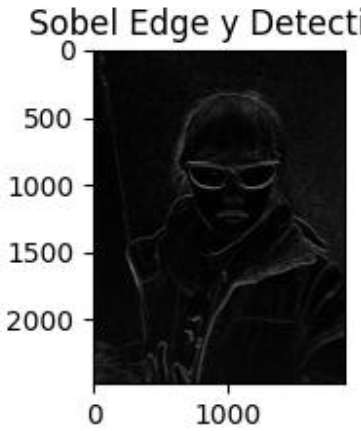
Sobel Gradient operator은 돌출된 값을 비교적 평균화하지만,수평과 수직에 놓여진 경계에 민감하게 반응한다는 단점이 있다. 코드는 아래와 같다.

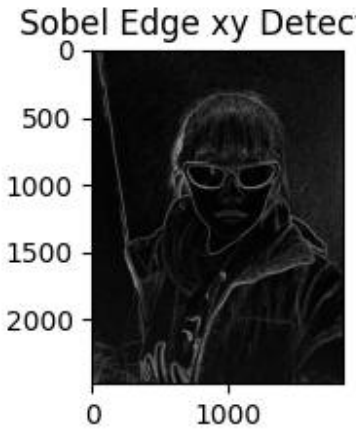
```
sobel_x = np.array([[-1, 0, 1],      //Sobel x filter
                    [-2, 0, 2],
                    [-1, 0, 1]])
sobel_y = np.array([[1, 2, 1],      //Sobel y filter
                    [0, 0, 0],
                    [-1, -2, -1]])
gradient_x = convolve2d(img_array, sobel_x, mode='same', boundary='symm')  //Sobel filter 적용
gradient_y = convolve2d(img_array, sobel_y, mode='same', boundary='symm')

sobel_intensity_x = np.sqrt(gradient_x ** 2)  //x방향 Edge 계산
sobel_intensity_y = np.sqrt(gradient_y ** 2)  //y방향 Edge 계산
sobel_intensity_xy = np.sqrt(gradient_x ** 2 + gradient_y **2)  //xy방향 Edge 계산
```

```
sobel_image_x = Image.fromarray(sobel_intensity_x)  //Edge 이미지로 변환해 저장
sobel_image_y = Image.fromarray(sobel_intensity_y)
sobel_image_xy = Image.fromarray(sobel_intensity_xy)

plt.subplot(2, 2, 1)   //원영상 시각화
plt.imshow(bw_image, cmap='gray')
plt.title('Original Image')

plt.subplot(2, 2, 2)   //x축 처리 영상 시각화
plt.imshow(sobel_image_x, cmap='gray')
plt.title('Sobel X Edge Detection')

plt.subplot(2, 2, 3)   //y축 처리 영상 시각화
plt.imshow(sobel_image_y, cmap='gray')
plt.title('Sobel Y Edge Detection')

plt.subplot(2, 2, 4)   //x축과 y축 처리 영상 시각화
plt.imshow(sobel_image_xy, cmap='gray')
plt.title('Sobel XY Edge Detection')

plt.tight_layout()
plt.show()
```









9. Laplace operator을 사용한 영상 처리 결과

Laplace operator은 Edge에서 부호가 바뀌는 것을 사용해 edge를 검출해내는 영상 처리 기법이다. edge 검출뿐만 아니라 노이즈 제거 등에도 사용된다. 공식은 다음과 같다.

$\nabla f^2(x,y) = f(x+1,y)+f(x-1),y)+f(x,y+1)+f(x,y-1)-4f(x,y)$ 이 공식을 배열로 표현하면,

[0 1 0] 이고, 이를 확장한 8 neighborhoods는 [1 1 1] 이다. 이를 적용한 코드는 아래와 같다.

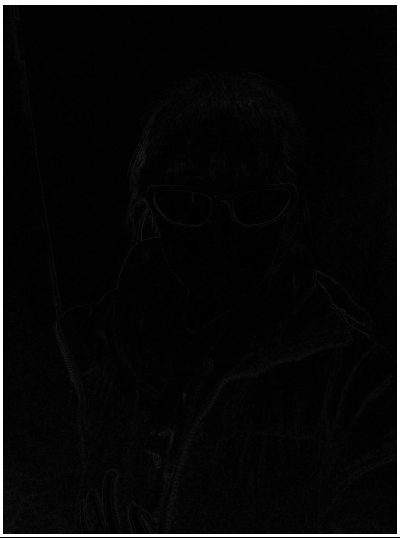
[1 -4 1] [1 -8 1]

[0 1 0] [1 1 1]

```
img_array = np.array(bw_image)     //이미지를 NumPy 배열로 변환
laplace4_kernel = np.array([[0, 1, 0],     //4 neighborhoods Laplace filter
                            [1, -4, 1],
                            [0, 1, 0]])
laplace8_kernel = np.array([[1, 1, 1],     //8 neighborhoods Laplace filter
                            [1, -8, 1],
                            [1, 1, 1]])

laplace4_output = convolve2d(img_array, laplace4_kernel,mode='same', boundary='symm')  //Laplace 적용
laplace8_output = convolve2d(img_array, laplace8_kernel, mode='same', boundary='symm')

laplace4_image = Image.fromarray(laplace4_output)  //Edge 이미지로 변환해 저장
laplace8_image = Image.fromarray(laplace8_output)
laplace4_image = laplace4_image.convert('L')    //흑백 이미지로 변환해 저장
laplace8_image = laplace8_image.convert('L')
laplace4_image.save('4 laplace_edge_image.jpg')    //이미지 파일로 저장
laplace8_image.save('8 laplace_edge_image.jpg')
```

<얼굴 원영상>

<4 neighborhoods>



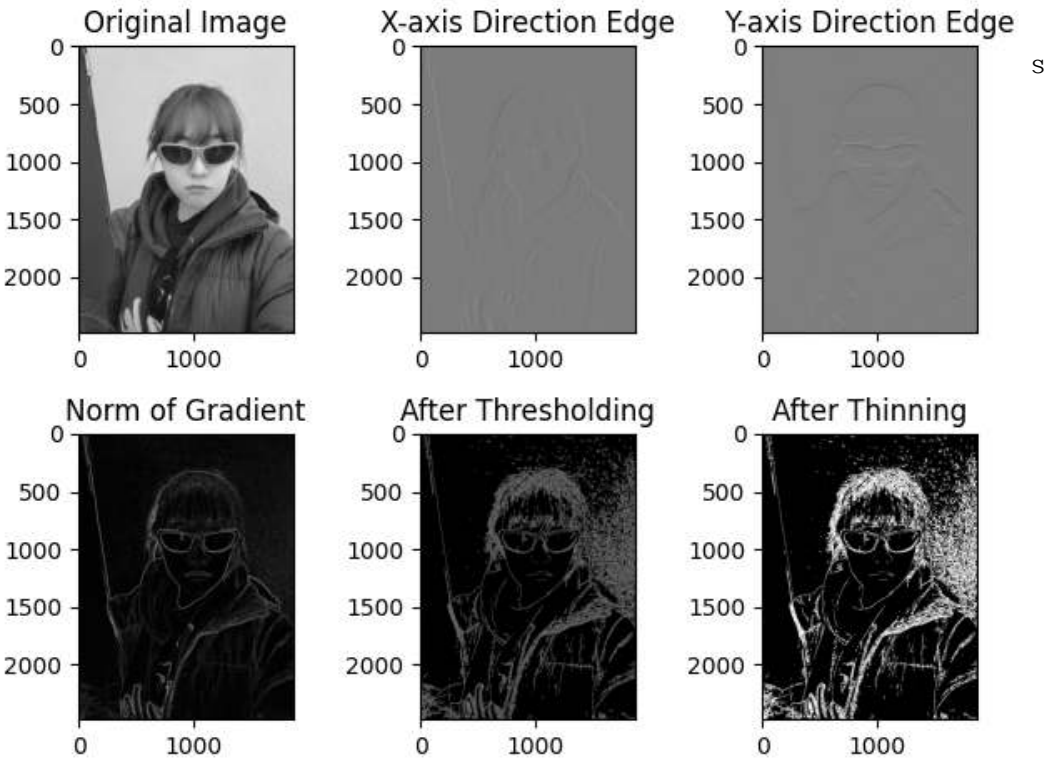
<8 neighborhoods>

이미지로는 잘 보이지 않지만 확대해보면 4 neighborhoods 영상보다 8neighborhoods 영상이 edge가 더 잘 검출된 것을 볼 수 있다.

10. Canny Edge Operator을 사용한 영상 처리 결과

Canny Edge Operator은 LoG의 계산 시간을 줄여주기 위해 LoG를 근사화한 영상 처리 기법이다. Gaussian mask로 영상을 회선한 뒤, 각각의 결과 픽셀들에서 기울기 요소를 계산하고, 각 화소에서 기울기 방향을 계산한다. 각 화소의 기울기 방향을 따라 2차 미분을 계산하고, 0 값의 위치를 찾는다. 0 값을 가진 점들을 Edge 점으로 분류해 Edge들을 검출한다. 강의록을 따라 원영상을 X-asis direction edge, Y-axis direction edge, Norm of Gradient, After Thresholding, After Thinning 과정으로 분류해 시각화해보았다. OpenCV를 사용했으며, 코드와 결과 영상은 다음과 같다.

```
import cv2   //OpenCV 라이브러리 사용
import matplotlib.pyplot as plt
import numpy as np

image = cv2.imread('faceimage.jpg', 0)  //이미지를 흑백으로 변환
smoothing_image = cv2.GaussianBlur(image, (5, 5), 0)  //Gaussian Smoothing Filter 적용

gradient_x = cv2.Sobel(smoothing_image, cv2.CV_64F, 1, 0, ksize=3)  //Sobel filter로 Edge 계산
gradient_y = cv2.Sobel(smoothing_image, cv2.CV_64F, 0, 1, ksize=3)
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_magnitude = np.uint8(gradient_magnitude)
gradient_direction = np.arctan2(gradient_y, gradient_x)

low_threshold = 50  //임계값 설정
high_threshold = 150
edges = cv2.Canny(gradient_magnitude, low_threshold, high_threshold)
kernel = np.ones((3, 3), np.uint8)   //Edge Thinning
thinned_edges = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)
```



s